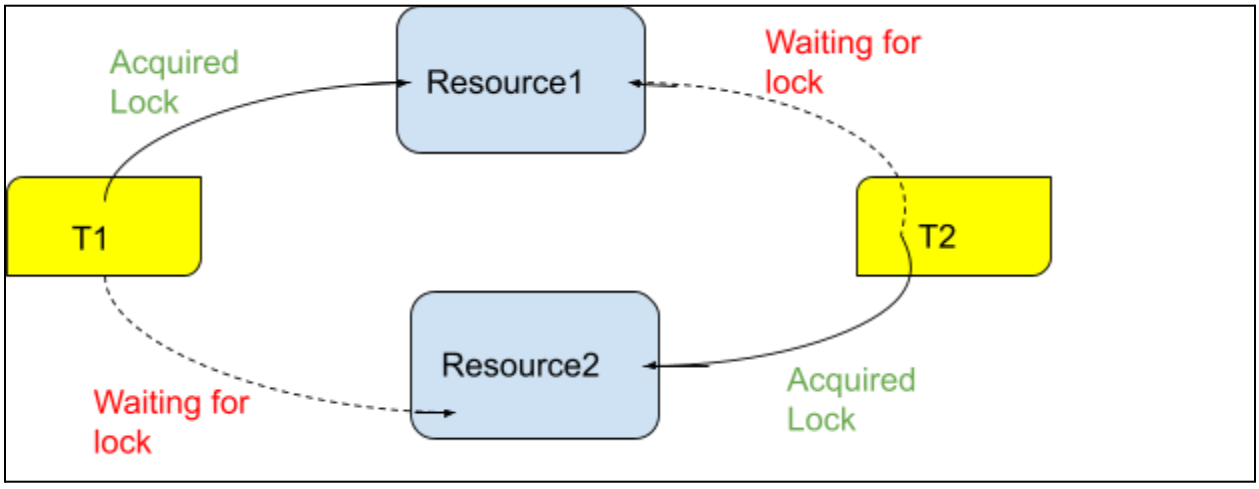
Advanced Java Week 3 - Day02

Agenda:

1. DeadLocks
2. Volatile
3. JAVA I/O API - Streams

DeadLocks

As discussed earlier DeadLocks are situations, wherein at least 2 threads are involved, for example, as illustrated in the following diagram, Thread T1 has acquired a lock on Resource R1 for Writing and Thread T2 has acquired a lock on Resource R2 for Writing



In parallel, The Thread T1 wants to acquire a lock on Resource2, which might not be achieved and similarly, The Thread T2 wants to acquire a lock on Resource1 , which is also not achieved until the locks are released on those resources by other Threads, this kind of a situation is termed as DeadLock.

Volatile :

Volatile keyword is used to indicate to the compiler that the corresponding data member on which the "Volatile" keyword is clamped, is available for access by multiple threads simultaneously.
Volatile keyword ensures that the data member is not cached in the stacks of the individual threads but instead it would be stored in the main memory.
It is another way of making class thread safe.

Can be applied on to primitive types or Objects, but volatile keyword cannot be applied to a method or a class

Therefore **volatile** keyword is just opposite to the keyword **"Synchronized"**, which allows only one thread to access the resource at any point of time.

Volatile therefore can be used in situations where multiple threads accessing a resource are not going to put the resource into jeopardy

For example In a Railway reservation System, Number of Tickets available for a destination being accessed by multiple threads simultaneously, may lead to inconsistency, therefore the threads which update the “No Of Tickets” available, should not be allowed to access this resource in parallel, therefore the activity through which the “NoOfTickets” is being accessed should be synchronized or in other words “NoOfTickets” should be part of something which is to be synchronized.

Whereas the Data **“Trains available”** for a particular destination can be accessible by multiple threads , may be for read operation, then this resource must be **“Volatile”,** but it should be noted that does not mean that all these threads can modify this data simultaneously leading this data into inconsistent state, it means that some other designated threads is modifying this data, which can be accessible by multiple threads possibly to read. Here too the group of threads which use a process to modify should be in the ambit of “synchronized” act, but other set of threads which have the responsibility of only accessing the current data may be for only read operation could be using this “volatile” data.

JAVA I/O API - Streams

Java provides a rich set of API under the package java.io , using which one can perform File Handling.

File Handling or the I/O operation usually used to represent Input/Output Operation over the files, can be implemented with the help of java.io package.

I/O operations are typically carried out on the Streams, which in turn facilitates the flow of this data into files.

“Streams” are flow of Data which are basically of 2 types **Input Stream** & **Output Stream**, Input Stream is the Stream which facilitates the flow of Data into the Application , the flow of data that is read from the file , whereas the Output Stream is the Stream which facilitates the flow of Data from the application into the file.
Hence, when we need to read from a file, we use InputStream and when we need to write we use OutputStream.
Java provides rich API to deal with these Streams, Streams are again categorized into 2 categories

1) Character Stream
2) Byte Stream

Character Streams allow the data flow in the character format and Byte Stream allows the data flow in the Binary format.
Depending upon the type of Data that we want to read/write , we can use the corresponding type of Stream, for example when we need to write/read textual data we can use the Character Streams & when we want to write/read some binary data for example audio, video or some image based data, then we can use Binary Stream.

These streams are also called Node Streams, these Node streams can be of the Type files, memory or thread based Node Streams.
In this chapter we are dealing with only File Streams.
The Classes provided by java.io package for file handling are as follows:

I/O Fundamentals (Contd.)

- The fundamental stream classes are:

Stream	Byte Streams	Character Streams
Source streams	InputStream	Reader
Sink streams	OutputStream	Writer

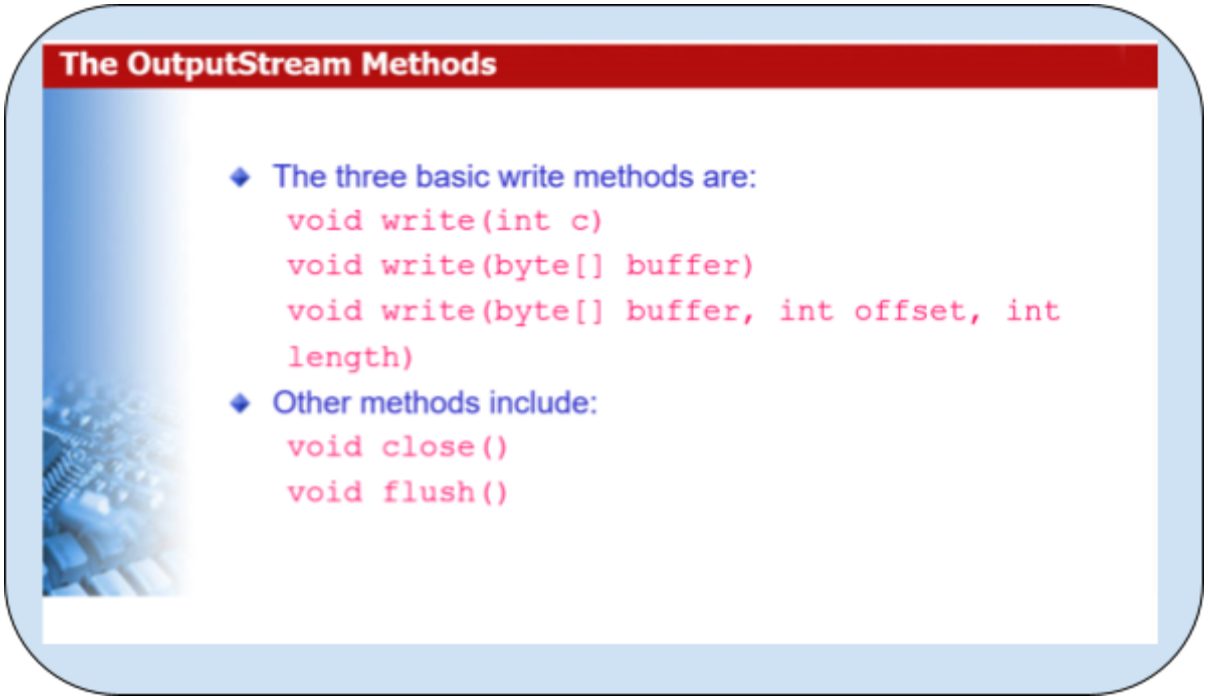
The InputStream Methods

- The three basic read methods are:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int
length)
```

- Other methods include:

```
void close()
int available()
long skip(long n)
boolean markSupported()
void mark(int readlimit)
```

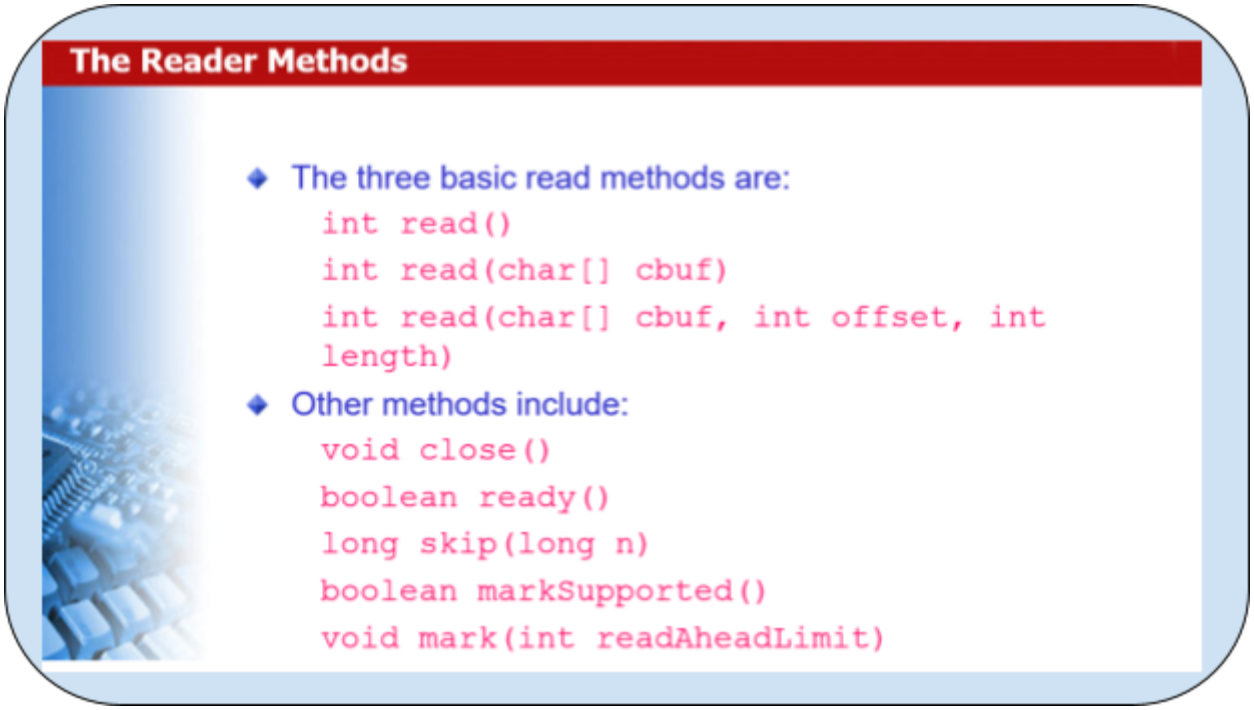


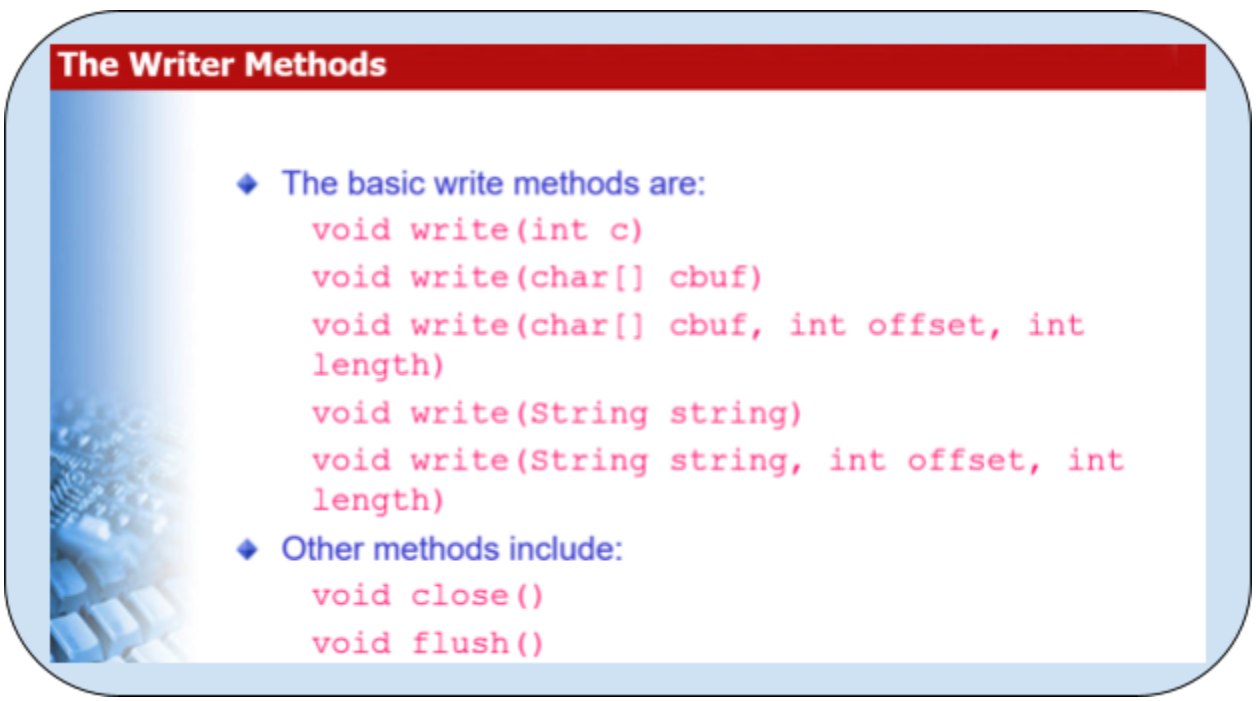
Above are the methods provided in InputStream and OutputStream classes to read & write from/to Binary Streams.

The Methods provided by the Reader & Writer classes to read & write from/to Character stream are as follows.

read() function returns numbers of bytes that are read from a file through Binary Stream
read(byte[] buffer) function reads and puts it into an array of bytes
read(byte[] buffer,int offset,int length) reads no. of bytes equal to length mentioned by length variable, from the offset position held by offset variable and puts it into an array of bytes

write(int c) : function writes no. of bytes into the file through binary stream
write(byte[] buffer) : function writes the byte[] array content into the file through binary Stream
write(byte[] buffer,int offset,int length): function writes the data present in array of bytes into the file from the location -offset until the length mentioned by length variable ,through binary stream





int read() method reads and returns the number of characters read from a file through a character Stream
int read(char[] buf) : method reads and returns the number of characters read from a file through a character stream and also puts the characters into an array of characters.
int read(char[] buf,int offset,int length): method reads and returns the number of characters read from a file through character stream and puts the characters into an array of characters from the file through character stream, it reads from the location held by the variable offset until the length mentioned by length variable

write(int c) : writes c number of characters into a file through character stream
write(char[] cbuf) : writes the characters passed through an array of characters cbuf into a file through character stream
write(char[] cbuf, int offset, int length) : writes the characters passed through an array of characters into the file through character stream, writes into the file from the location identified by the variable offset until the length specified by the length variable.

close () method closes the corresponding stream (InputStream / OutputStream)
ready() method return true/false based on the context, if there are characters that are ready to be read
skip(long n) method skips n number of characters and **reads** remaining number of characters and returns that number
markSupported() : method returns boolean value indicating whether the marking of characters is possible or not on the character stream
mark(int limit) : marks the specified number of characters in the file for reading
flush() : function flushes the implicit buffer, before writing into a file

—----------EOD----------